Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
TRITA-EECS-AVL-2020:4 SE-10044 Stockholm
ISBN 100- Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
   `https://doi.org/10.1145/3397537.3397567`

2. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Network and Distributed System Security Symposium (NDSS 2021), MADWeb
   `https://doi.org/10.14722/madweb.2021.23004`

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
   `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

4. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023
   `https://www.sciencedirect.com/science/article/pii/S0167404823002067`

5. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
   `https://doi.org/10.1145/3358504.3361228`

ACKNOWLEDGEMENT

ACRONYMS

List of commonly used acronyms:

Wasm WebAssembly

Contents

Part I

Thesis

01 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: Mapping of the contributions to the research papers appended to this thesis.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

2.1.1 WebAssembly toolchains

TODO Mention, stress the landscape of tools that involve Wasm. Include analysis tools, fuzzers, optimizers and malware detectors.
TODO End up motivating the need of Software Diversification for: testing and reliability.

2.2 Software diversification

2.3 Generating Software Diversification

2.3.1 Variants generation

2.3.2 Variants equivalence

2.4 Exploiting Software Diversification

2.4.1 Defensive Diversification

2.4.2 Offensive Diversification

03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

This dissertation makes contributions to the field of Software Diversification for WebAssembly with three main technical contributions: CROW, MEWE, and wasm-mutate. In the following text we detail each one of these contributions.

3.1 Generating Software Diversification for WebAssembly

WebAssembly programs are obtained ahead of time. This means that the source code is compiled to Wasm binaries before they come to host engines to be compiled to machine code and then executed. The process of obtaining a WebAssembly program can be roughly separated in three high-level components: source code, compiler and the generated WebAssembly program. Software Diversification can be applied to any of these three main components. Yet, applying diversification at the source code level is not practical since it will limit such approach to specific programming languages, e.g. one diversifier per possible language compiling to WebAssembly. Therefore, this thesis focus on the other two components, the compiler and the generated WebAssembly program itself. In Figure 3.1 we show the landscape of our approaches. They focus on two main strategies: compiler-based and binary-based. The compiler-based approaches are highlighted in red and green, while the binary-based approaches are highlighted in blue.

Our compiler-based approaches are illustrated in Figure 3.1 as the red and green squared tooling. The work of Hilbig et al. [**?**] in 2021 highlights that 70% of the Wasm binaries in the wild are created with LLVM-based compilers. Based on such analysis, we provide artificial software diversity for Wasm as a compiler-based approach through LLVM. LLVM is a compound of three main components [**?**]. First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation processes improve the LLVM IR. Third and final, the backend component is in charge of generating the target Instruction Set. We alter the LLVM pipeline that compiles source code to Wasm by introducing a diversifier component. The diversifier generates LLVM IR variants from the output of the frontend. The diversifier and the custom Wasm LLVM backend compose CROW, which creates Wasm program variants out of a source code program. In addition, an orthogonal tool comes from the generation of LLVM IR variants, MEWE [?], which merges and creates multivariant binaries to provide Multivariant Execution for Wasm.

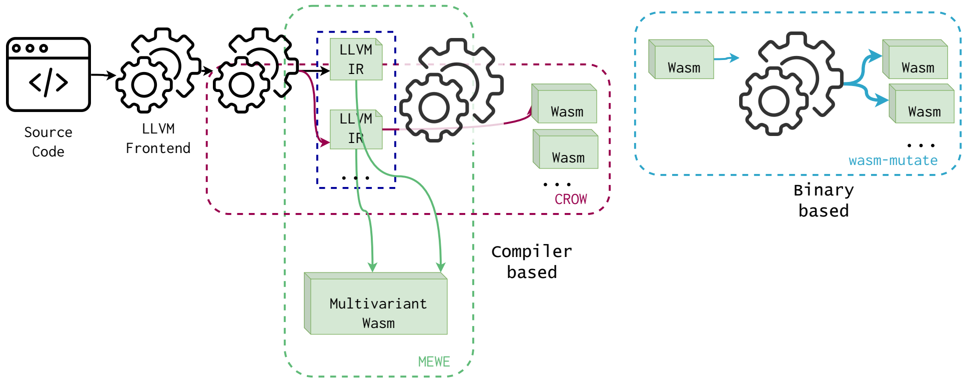

Figure 3.1: Approach landscape.

■ 3.2 CROW: Code Randomization of WebAssembly

This section describes the red squared tooling in Figure 3.1 named CROW [?]. CROW is a tool tailored to create semantically equivalent Wasm variants from an LLVM front-end output. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

In Figure 3.2, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function in its input LLVM. For all LLVM instructions, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into Wasm binaries. In the following text, we describe our design decisions.

■ 3.2.1 Variants' generation

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier of Jacob et al. [?]. A superoptimizer focuses on

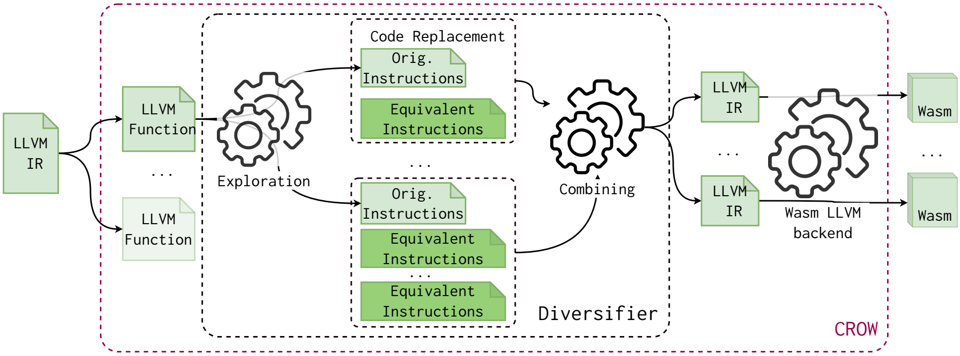

Figure 3.2: CROW components following the diagram in ??. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

searching for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [?] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the superdiversifier idea of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on handwritten rules **TODO** This is contradictory to our binary solution. . Concretely, we can control the quality of the generated codes. Besides, CROW always generates equivalent programs because it is based on a solver to check for equivalence. Second, there is a battle-tested superoptimizer for LLVM, Souper [?]. This latter makes it feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger that a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of the commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

3.2.2 Constant inferring

One of the code transformation strategies of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in **??**.

After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n", current)
            ;
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;



    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1].

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the Wasm backend. We disable all optimizations in the Wasm backend that could reverse the CROW transformations. In the following enumeration, we list three concrete optimizations that we remove from the Wasm backend.[2]

- Constant folding: this optimization calculates the operation over two (or more) constants in compiling time, and replaces the original expression by its constant result. For example, let us suppose $a = 10 + 12$ a subexpression to be compiled, with the original optimization, the Wasm backend replaces it by $a = 22$.
- Expressions normalization: in this case, the comparison operations are normalized to its complementary operation, e.g. $a > b$ is always replaced by $b <= a$.
- Redundant operation removal: expressions such as the multiplication of variables by $a = b2^n$ are replaced by shift left operations $a = b << n$.

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in **??**.

[2]We only illustrate three of the removed optimization for the sake of simplicity.

3.2.3 CROW instantiation

Let us illustrate how CROW works with the simple example code in ??. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of ??. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of ?? shows.

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of ?? shows. CROW synthesizes $2+1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of ??. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as ?? shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as ?? illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of ?? illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*Software!Replacement overlapping.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformation only to the instruction for which it was found in the first place. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

3.3 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). The tool generates application-level multivariant binaries without changing the operating system or Wasm runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step 2 in **??** shows. CROW generates each one of these variants with fine-grained diversification at the instruction level, applying the majority of the strategies discussed in **??** and *constant inferring*. MEWE adds a new mutation strategy. It inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE from the blue highlighted square in **??**. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant Wasm binary using the same custom Wasm LLVM backend from CROW. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

3.3.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE

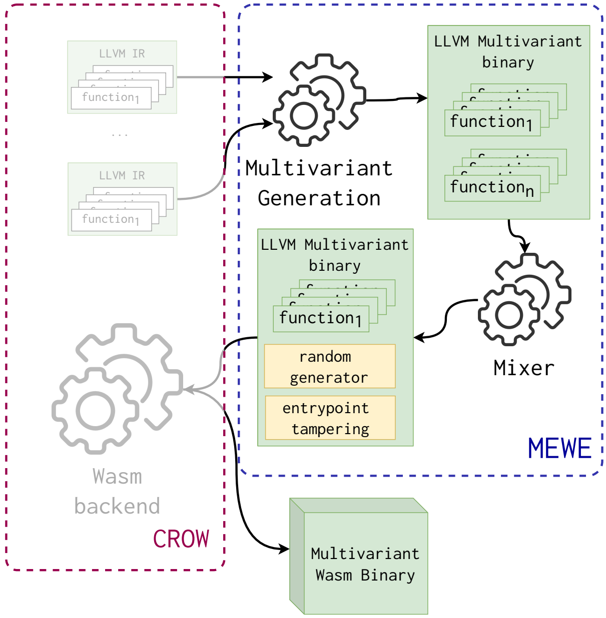

Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed.

(bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In Listing 3.3, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature

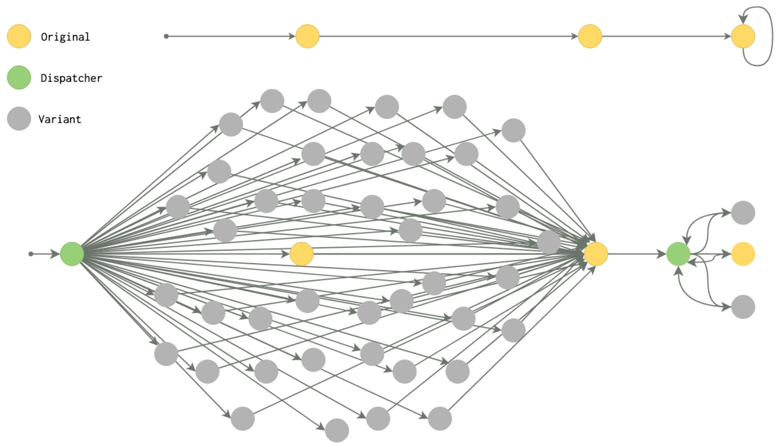

Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow).

is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again. Here we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.3: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

3.3.2 The Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[3] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

3.4 Wasm-mutate

TODO Motivate **TODO** What happens with the other 30% of the binaries?

3.4.1 Variants' generation

TODO The egraph thingy

3.5 Discussion

TODO Comparison of the approaches

[3] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

04 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

4.1 Offensive Software Diversification

4.1.1 **Use case 1:** Automatic testing and fuzzing of WebAssembly consumers

TODO We explain the CVE. Make the explanation around "indirect memory diversification"

4.1.2 **Use case 2:** WebAssembly malware evasion

TODO The malware evasion paper

4.2 Defensive Software Diversification

4.2.1 **Use case 3:** Multivariant execution at the Edge

TODO Disturbing of execution time. Go around the web timing attacks.

4.2.2 **Use case 4:** Speculative Side-channel protection

TODO Go around the last paper

05 CONCLUSIONS AND FUTURE WORK

REFERENCES

[1] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," *Proceedings of the Web Conference 2021*, 2021.

[2] LLVM, " The LLVM Compiler Infrastructure ." `https://llvm.org/`, 2003.

[3] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[4] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[5] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[6] M. Henry, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Nov 1987.

[7] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[8] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[9] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[10] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb

`https://doi.org/10.14722/madweb.2021.23004`

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

https://www.sciencedirect.com/science/article/pii/S0167404823002067

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228